Quantum Programming Languages

An Overview

Christopher Baumgartner-Trösch

Supervisor: Univ.-Prof. Dr. Georg Moser

SE Specialisation Seminar SS 2024
Seminar lecturer: assoz. Prof. Dr. Cezary Kaliszyk
Institute for Computer Science
University of Innsbruck

June 13, 2024

This report provides an overview of quantum programming languages, emphasizing their theoretical foundations and practical applications. The background and significance of quantum computing are discussed, highlighting the advantages of quantum algorithms and the interdisciplinary nature of this emerging technology. The report introduces fundamental quantum computing concepts such as qubits, superposition, and entanglement, which are necessary for understanding quantum algorithms.

A detailed examination of various quantum programming languages is presented, with a particular focus on the Quantum Computation Language (QCL). The syntax, structure, and features of QCL are explored. Additionally, the report discusses the Common Quantum Assembly Language (cQASM), which addresses the need for a standardized, hardware-independent language to enhance interoperability and reliability across different quantum computing platforms.

To illustrate practical applications, examples using Qiskit, an open-source quantum computing framework developed by IBM, are provided. These examples demonstrate the construction, execution, and visualization of quantum circuits.

The report concludes by highlighting the importance of quantum programming languages for harnessing the full potential of quantum computers. As quantum hardware continues to advance, the development and standardization of robust quantum programming languages and tools will be increasingly relevant for the progression of quantum computing.

1 Introduction

1.1 Background and Significance of Quantum Computing

The advent of quantum computing marks a significant change in computational technology. It uses quantum mechanics to perform tasks that classical computers cannot handle efficiently. Classical computers use bits as their basic unit of information, which can be either 0 or 1. Quantum computers, however, use quantum bits or qubits, which can exist in multiple states simultaneously due to superposition. This allows quantum computers to process large amounts of information in parallel, providing a significant speed-up for certain computations.

The objective of quantum computing is to address complex problems that are challenging for classical computers. These include factoring large numbers, simulating quantum physical processes, optimizing large systems, and solving combinatorial problems. Quantum algorithms, such as Shor's algorithm for factoring and Grover's algorithm for searching, offer significant advantages over classical algorithms.

Additionally, quantum computing has potential applications in fields such as cryptography, where it can break widely-used encryption schemes, and material science, where it can accurately simulate molecular structures. This makes quantum computing a highly anticipated technology with implications for various industries, including pharmaceuticals and logistics.

The significance of quantum computing goes beyond its technical capabilities. It brings together multiple scientific disciplines, such as physics, computer science, and engineering, encouraging interdisciplinary collaboration and innovation. As research advances and quantum hardware improves, the need for standardized programming languages and tools, like those discussed in this report, becomes essential.

1.2 Objectives of the report

This report provides an overview of quantum programming languages, focusing on their theoretical foundations, practical applications, and current state of development. By examining both imperative and functional paradigms, it explains the different approaches and their respective strengths and weaknesses.
The initial section covers fundamental quantum computing concepts such as qubits, superposition, and entanglement, which are essential for understanding quantum algorithms. The Quantum Computation Language (QCL) is discussed, with regards to its syntax, structure, and (quantum) features.
Another focus is the Common Quantum Assembly Language (cQASM), which aims to unify QASM's many dialects to standardize the representation of quantum circuits.
Finally, a practical example using Qiskit is given to demonstrate how to construct, execute, and visualize quantum circuits.

2 Theoretical Foundations of Quantum Computing

2.1 Qubits and Superposition

Quantum bits (qubits) represent the fundamental unit of quantum computing, differing significantly from their classical counterparts in terms of operational principles. Qubits are derived from subatomic particles, such as photons, electrons, neutrons, or atoms, which possess quantum mechanical properties. In contrast to classical bits, which are binary and exist in a state of either 0 or 1, qubits employ the principle of superposition, enabling them to exist in multiple states simultaneously. This property is represented mathematically as:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \quad \alpha, \beta \in \mathbb{C}^2 \tag{1}$$

The ket symbol $|\psi\rangle$ represents the state of the qubit, while the complex numbers α and β indicate the probabilistic amplitudes of the qubit being in the states $|0\rangle$ and $|1\rangle$, respectively. The measurement of the state of a qubit results in the superposition $|\psi\rangle$ collapsing to either $|0\rangle$ or $|1\rangle$. The distribution of probabilities between α and β is described as

$$\|\alpha\|^2 + \|\beta\|^2 = 1. \tag{2}$$

Consequently, the probability of a qubit being in state α (or β) is $\|\alpha\|^2$ (or $\|\beta\|^2$). This property is the fundamental principle of quantum computing, which distinguishes it from classical computing by enabling simultaneous computation across multiple states.

The two initial states can be visualized in the following way:

$$|\psi\rangle = |0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{or} \quad |\psi\rangle = |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad |\psi\rangle \in \mathbb{C}^2. \tag{3}$$

To represent states in superposition (1), it is sufficient to write: $\begin{pmatrix} \alpha \\ \beta \end{pmatrix}$

Thus, we see that $|0\rangle$ and $|1\rangle$ are also themselves described by α and β with

$$\begin{pmatrix}\alpha = 1\\ \beta = 0\end{pmatrix} \quad \text{and} \quad \begin{pmatrix}\alpha = 0\\ \beta = 1\end{pmatrix}, \text{ respectively.} \tag{4}$$

These initial states can be manipulated by quantum gates that perform quantum operations.

The state of a quantum system with 2 qbits can be described in the following way [1, p. 25]:

$$|\Psi\rangle = \alpha|00\rangle + \beta|10\rangle + \gamma|01\rangle + \delta|11\rangle \quad \text{with} \quad |\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1$$

$$|00\rangle = \begin{pmatrix}1\\0\\0\\0\end{pmatrix}, \quad |01\rangle = \begin{pmatrix}0\\1\\0\\0\end{pmatrix}, \quad |10\rangle = \begin{pmatrix}0\\0\\1\\0\end{pmatrix}, \quad |11\rangle = \begin{pmatrix}0\\0\\0\\1\end{pmatrix} \quad \in \quad \mathbb{C}^4. \tag{5}$$

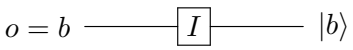
2.2 Entanglement

Entanglement is an inherent property of qubits. It enables qubits to establish a correlation such that the measurement or manipulation of one qubit affects the states of the other entangled qubits. When qubits are entangled, their states become interdependent, allowing the state of one qubit to be inferred by measuring another, regardless of the distance between them.
To illustrate, if two qubits are initially entangled and subsequently separated, measuring the state of one qubit (e.g., determining it to be in the state $|1\rangle$) will immediately determine the state of the other qubit (e.g., $|0\rangle$). This phenomenon occurs instantaneously, even over large distances, and is fundamental to quantum communication and computation protocols [2, p. 6].

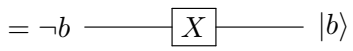
2.3 Quantum Gates & Operations

2.3.1 Quantum Gates

As an intuition, the following quantum gates and their representations based on [3, ch. 6] will be introduced. The diagrams are to be read from right (input state) to left (output state o).

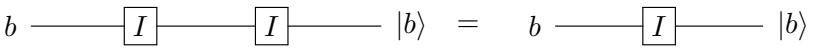


(6)

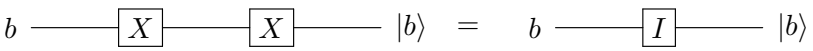


$b \in \{0, 1\}$

In this context, the symbol I represents the identity gate, while the symbol X represents the bit flip gate, which transforms the state vector $|0\rangle$ into the state vector $|1\rangle$ and vice versa. It should be noted that the operations are reversible by themselves, in the sense that two consecutive gates of the same type are equivalent to the identity operation:



(7)



Of greater interest than these quantum counterparts of classical gates is the Hadamard gate, which puts the qubit into superposition. At the point of measurement, the qubit's value collapses either to $|0\rangle$ or $|1\rangle$ with uniform probability.

$$o = \begin{cases} 0 & \text{with } \mathbb{P}(0) = \frac{1}{2} \\ 1 & \text{with } \mathbb{P}(1) = \frac{1}{2} \end{cases} \qquad o \text{ —[H]— } |b\rangle \qquad b \in \{0, 1\}$$

It is noteworthy that the Hadamard gate is also invertible by itself, without loss of information about the initial state of the qubit. This phenomenon appears to be intuitive from a classical perspective. Prior to measurement, the qubit remains correlated with the input state; only the measurement erases and resets its state to a classical, non-superpositional state. Provided that no measurement is performed between the two Hadamard gates, the initial state can be reverted to:

$$b \text{ —[H]—[H]— } |b\rangle \quad = \quad b \text{ —[I]— } |b\rangle$$

Another gate is the sign flip gate (which should not be confused with the bit flip gate). This gate behaves like the identity gate and is therefore invertible by itself:

$$o = b \text{ —[Z]— } |b\rangle \qquad b \in \{0, 1\}$$

$$b \text{ —[Z]—[Z]— } |b\rangle \quad = \quad b \text{ —[I]— } |b\rangle$$

The Z gate transforms the basis states from $|0\rangle$ and $|1\rangle$ to $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$, respectively (hence 'sign-flip' gate). This and the subsequent application of the Hadamard gate causes the Z gate to act like an X gate, which flips the states $|0\rangle$ and $|1\rangle$. This demonstrates that $Z \neq I$.

$$b \text{ —[H]—[Z]—[H]— } |b\rangle \quad = \quad b \text{ —[X]— } |b\rangle$$

3 Quantum Operations

3.1 Matrix Multiplications for Quantum Operators

The linear nature of quantum mechanics allows for the expression of quantum operators using matrix multiplications. In the formalism of quantum mechanics, states are represented by vectors in a complex Hilbert space, and quantum operators are represented by matrices that act on these state vectors. For a quantum state $|\psi\rangle$, an operator U applied to $|\psi\rangle$ is expressed as:

$$|\psi_{\text{final}}\rangle = \psi_{\text{final}} = U\psi = U \begin{pmatrix} \psi_0 \\ \psi_1 \end{pmatrix} = \begin{pmatrix} U_{0,0} & U_{0,1} \\ U_{1,0} & U_{1,1} \end{pmatrix} \begin{pmatrix} \psi_0 \\ \psi_1 \end{pmatrix} = \begin{pmatrix} U_{0,0}\psi_0 + U_{0,1}\psi_1 \\ U_{1,0}\psi_0 + U_{1,1}\psi_1 \end{pmatrix} \in \mathbb{C}^2.$$

In this context, $|\psi\rangle$ represents a quantum state, which is expressed as a column vector. The unitary operator U is represented by a matrix. The final state $|\psi_{\text{final}}\rangle$ is obtained by multiplying the matrix U with the state vector ψ. This matrix multiplication effectively transforms the state in accordance with the rules of the quantum operation defined by U.

3.2 The Bloch Sphere

The pure state space of a single qubit is represented geometrically by the Bloch sphere. In three-dimensional space, a point on the surface of a unit sphere can represent any pure state of a qubit. An intuitive approach to viewing the state of a qubit and the results of quantum gates is through a Bloch sphere.
A general qubit state $|\psi\rangle$ can be written as:

$$|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle$$

where the spherical coordinates are φ and θ. In this case, the unique determinants of the point on the Bloch sphere that corresponds to the qubit state are the polar angle, θ, and the azimuthal angle, φ. θ is the angle from the positive z-axis and ranges from 0 to π. φ is the angle from the positive x-axis in the x-y plane and ranges from 0 to 2π [2].

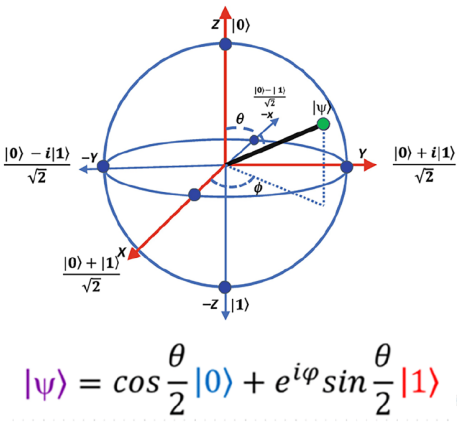


Figure 1: Bloch Sphere [4]

In three-dimensional space, the coordinates (x, y, z) on the Bloch sphere can be derived as follows:

$$x = \sin\theta\cos\varphi$$
$$y = \sin\theta\sin\varphi$$
$$z = \cos\theta$$

The Bloch sphere is three-dimensional because it captures the complex amplitudes of the qubit state in a real three-dimensional space. The representation ensures that any unitary transformation (quantum gate) acting on the qubit can be visualized as a rotation of the Bloch vector on this sphere.

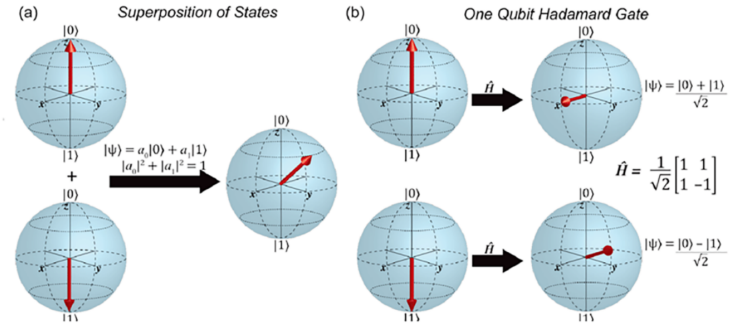


Figure 2: Bloch sphere representation of Hadamard gate [2]

4 Classification of Quantum Programming Languages

Quantum programming languages can be classified into several categories based on their operational paradigms and underlying models. In [5] a comprehensive taxonomy of these languages is provided. They can be divided primarily into imperative, functional, and other paradigms, which often include mathematical formalisms not intended for direct computer execution.

4.1 Imperative Quantum Programming Languages

Imperative quantum programming languages are constructed upon classical imperative paradigms, utilising statements to modify the global state of the system. These languages frequently incorporate features from classical languages such as C and Java. An early and notable example of an imperative quantum programming language is QCL (Quantum Computation Language), developed by Bernhard Ömer between 1998 and 2003, which will be discussed in the following section. QCL utilizes a syntax derived from C and provides a classical programming sublanguage along with high-level quantum programming features such as automatic memory management and user-defined operators and functions [5, p. 68].

Another notable quantum programming language is the one developed by Betelli and colleagues. This language, based on C++, incorporates a library for quantum operations, enabling the runtime construction and optimization of quantum operators [5, p. 68].

Sanders and Zuliani introduced qGCL (quantum Guarded Command Language), derived from Dijkstra's guarded command language, which focuses on the derivation and verification of quantum algorithms [5, p. 68].

LanQ, created by Mlnarik, is an imperative quantum programming language with a C-like syntax. It supports both classical and quantum process operations, including process creation and interprocess communication, and features a simulator. These languages provide various tools for the development and verification of quantum algorithms. [5, p. 68].

4.2 Functional Quantum Programming Languages

Functional quantum programming languages do not rely on changing the global state but perform mathematical transformations through the execution of functions. These languages are often based on lambda calculus, which investigates the properties of functions and their computability. An example is QPL (Quantum Programming Language), proposed by Peter Selinger, which represents programs via functional flow charts and textual syntax, integrating both classical and quantum data within the same formalism [5, p. 68].
QML, developed by Altenkirch and Grattage, is another functional quantum programming language. It is based on linear logic and is designed to manage quantum states without duplication [5, p. 69].

5 QCL: Quantum Computation Language

The Quantum Computation Language (QCL) is a high-level programming language that has been specifically designed to assist those with experience in classical computer programming to transition into the field of quantum computing. In contrast to traditional programming languages, QCL is designed to be independent of any specific quantum hardware. The syntax and procedural structure of QCL are heavily influenced by established programming languages such as C and Pascal, which flattens the learning curve for those familiar with these languages.

QCL enables the comprehensive implementation and simulation of quantum algorithms, while also accommodating the necessary classical components. This dual capability allows programmers to write and simulate entire quantum programs within a single unified formalism. Consequently, QCL serves not only as a practical tool for current quantum computing applications but also as a bridge for greater interaction between classical computing and quantum algorithm development [6, p. 16].

5.1 Models of Quantum Computation

In classical information theory, the concept of the universal computer can be represented by several equivalent models corresponding to different scientific approaches. [1, p. 36].

Classical Model	**Quantum Model**
Partial Recursive Functions	Unitary Operators
Turing Machine	Quantum Turing Machine (QTM)
Logical Circuit	Quantum Gates
Universal Programming Language	Quantum Programming Languages (QPLs)

Table 1: Classical and quantum computational models [1, p. 36]

5.1.1 Quantum Turing Machines

Quantum Turing Machines (QTM) have been proposed in several ways as models of a universal quantum computer. A superposition of basic states $|l, j, s\rangle$, where l is the internal state of the head, j is the head position, and s is the binary representation of the tape content, represents the entire machine state $|\Psi\rangle$. The infinite bit string s must meet the zero tail state condition in order to preserve a separable Hilbert space; this means that only a limited number of bits can have $s_m \neq 0$.The step operator T is the quantum counterpart of the transition function in a classical probabilistic Turing machine. It needs to be unitary in order to guarantee the existence of a corresponding Hamiltonian and to satisfy locality requirements for the head and tape qubit that are affected.
Several proposals have been made for Quantum Turing Machines (QTM) as models of a universal quantum computer. The complete machine state $|\Psi\rangle$ is represented by a superposition of base states $|l, j, s\rangle$, where l is the head's internal state, j is the head position, and s is the binary representation of the tape content. To maintain a separable Hilbert space, the infinite bit string s must satisfy the zero tail state condition, allowing only a finite number of bits where $s_m \neq 0$. The quantum equivalent of the transition function in a classical probabilistic Turing Machine is the step operator T, which must be unitary to ensure the existence of a corresponding Hamiltonian and to meet locality conditions for the affected tape qubit and head movement. Although QTMs provide a means of measuring execution times, running-time complexity, that is, the amount of T applications relative to the problem size, remains a challenge, as does choosing the right step operator. QTMs have limited significance beyond quantum complexity theory [1, p. 37].

5.1.2 Quantum Circuits

Quantum circuits are the quantum computing analog of classical Boolean feed-forward networks, with the primary distinction being that all quantum computations must be unitary, permitting evaluation in both directions. Quantum circuits consist of elementary gates and operate on qubits, making the dimensionality of the Hilbert space $H = 2^n$, where n is the number of qubits.

Compared to classical Boolean feed-forward networks, quantum circuits have specific constraints:

- Only n-to-n networks are allowed, meaning the total number of inputs must equal the total number of outputs.
- Only n-to-n gates are permitted.
- Forking of inputs is prohibited, as qubits cannot be copied (the no-cloning theorem).
- No "dead ends" are permitted, as erasure of a qubit is not a unitary operation.

A universal set of elementary gates must be available to implement all conceivable unitary transformations. Composite gates can then be built from this set. Unlike QTMs, the complexity of the problem is explicitly reflected in the number of gates needed to implement it, since the gate notation is intrinsically constructive [1, p. 37-38].

5.2 The QCL Interpreter - Enabling Quantum Simulations

The Quantum Computation Language (QCL) features the QCL interpreter, which is capable of simulating quantum computing environments. This interpreter is of great use for developers and researchers, allowing them to execute and interact with quantum algorithms in a controlled, simulated setting. Its ability to dynamically adjust the number of qubits and operate in an interactive shell environment with real-time debugging capabilities makes the QCL interpreter a versatile tool for testing quantum code.
The interpreter includes a numerical simulator that supports detailed simulation of quantum operations. This is relevant if programmers want to understand the effects of their quantum programs in detail, and it also comes in handy for educational purposes since step-by-step execution can be an aid in learning.
These features make the QCL interpreter useful in the practical application of quantum algorithms and in advancing the understanding and development of quantum computing solutions. It provides a robust platform for developing, executing, and analyzing quantum programs [6, p. 18].

5.2.1 Syntax and Structure

QCL programs are defined using a context-free LALR(1) grammar, which ensures straightforward and predictable parsing and compilation behaviors. The syntax includes expressions, statements, and definitions, organized in a structured format that can be read dynamically from files or directly from command line inputs. [6, p. 20-21].

5.2.2 Statements

Statements in QCL are executed as they are encountered, including direct commands, procedure calls, and more complex control structures. The language supports conditional execution and looping constructs familiar to classical programmers. For example, conditional printing based on random outcomes demonstrates the integration of quantum randomness in decision-making processes [6, p. 21].

5.2.3 Definitions and Expressions

In QCL, definitions bind operations or values to identifiers, which can then be used throughout a program. These include variable and constant definitions as well as more complex routine definitions [6, p. 21].

Expressions in QCL can be complex constructs composed of literals, variable references, subexpressions, operators, and function calls.

An example provided in the documentation of QCL illustrates how factorial calculations can be expressed in QCL. It also demonstrates recursion [6, p. 21]:

```
qcl> print "5 out of 10:", fac(10) / fac(5) ^ 2, "combinations."
: 5 out of 10: 252 combinations.
```

5.3 Classical Features

Type	Description	Examples
`int`	integer	`1234, -1`
`real`	real number	`3.14, -0.001`
`complex`	complex number	`(0,-1), (0.5, 0.866)`
`boolean`	logic value	`true, false`
`string`	character string	`"hello world", ""`

Figure 3: QCL's Classical Types and Literals [6].

5.3.1 Constant Expressions

QCL supports various classical data types, including integers, real numbers, complex numbers, booleans, and strings. These types are essential for constructing classical expressions in QCL, allowing for a wide range of arithmetic and logical operations. Each data type has a specific syntax: integers are derived from the C signed long type, reals from double, and complex numbers are represented as pairs of reals for the real and imaginary parts [6, p. 22-23].

5.3.2 Operators

Operators in QCL include arithmetic, comparison, and logic operators. Arithmetic operators handle basic mathematical functions and support operations across different data types, automatically converting them to the most general type applicable. Comparison operators return boolean values based on the comparison of operands, while logic operators facilitate the construction of complex logical expressions [6, p. 23-25].

5.3.3 Functions

QCL provides built-in functions for trigonometric calculations, exponentiation, logarithms, and operations on complex numbers. These functions are overloaded to accept multiple argument types and return the appropriate type based on the input [6, p. 25-26].

5.4 Quantum Features

5.4.1 Quantum Registers

Qubits, which together form the quantum machine state $|\Psi\rangle$, are used in the QCL to represent the memory structure of a quantum computer. Due to the destructive nature of measurements, this state resides as a vector in the Hilbert space $H = \mathbb{C}^{2^n}$ and cannot be viewed directly [**?**, p. 30-31]oemer1998qcldoc.

In QCL, quantum registers serve as a conduit between the controlling algorithm and the state of the quantum machine. Except for reset commands, they serve as pointers to sequences of individual qubits, and quantum registers are used for all operations on the machine state. $\frac{n!}{(n-m)!}$ distinct m-qubit registers are feasible with an n-qubit quantum computer, which means that $\frac{n!}{(n-m)!}$ possible operations on the machine state are possible for each unitary or measurement operation on an m-qubit register [**?**, p. 30]oemer1998qcldoc.

5.4.2 Quantum Expressions

In QCL, quantum expressions are references that point to qubits or registers used in quantum operations, either as arguments for quantum operators or to declare specific references. The types of quantum expressions are described in the following [6, p. 35-36]:

Single qubits are identified by an index, allowing targeting of individual qubits within a register for specific operations.

A substring is a subregister specified by the starting qubit's offset and either the ending qubit's offset or the total length of the subregister. This makes manipulation of subsets of qubits within a larger register easier.

QCL also supports the creation of combined registers using the concatenation operator &, which merges multiple registers into one, provided there is no overlap among the qubits involved. This functionality increases the flexibility of the language, making it easier for programmers to dynamically combine and manipulate multiple qubit registers.

Expr.	**Description**	**Register**
`a`	reference	$\langle a_0, a_1 \dots a_n \rangle$
`a[i]`	qubit	$\langle a_i \rangle$
`a[i:j]`	substring	$\langle a_i, a_{i+1} \dots a_j \rangle$
`a[i\l]`	substring	$\langle a_i, a_{i+1} \dots a_{i+l-1} \rangle$
`a&b`	concatenation	$\langle a_0, a_1 \dots a_n, b_0, b_1 \dots b_m \rangle$

Figure 4: QCL's Quantum Expressions [6]

6 cQASM: Common Quantum Assembly Language

There is an increasing need for a standardized, hardware-independent method for representing quantum circuits. The Common Quantum Assembly Language (cQASM) addresses this. cQASM, proposed by Khammassi et al., provides a universal intermediate representation that promotes interoperability among different quantum compilation and simulation tools while abstracting the specifics of various quantum hardware technologies [7, p. 1].

Quantum Assembly Language (QASM) is commonly used as an intermediate representation to describe quantum circuits. However, many QASM dialects have emerged, each tailored to specific tools or hardware, which has lead to fragmentation. This requires multiple translators and risks information loss during translations. cQASM aims to unify these dialects by standardizing the language. This guarantees reliable translation between high-level quantum algorithms and hardware-specific code [7, p. 1].

The main goal of cQASM is to provide a common standard for quantum computing tools and to amplify the development and execution of quantum algorithms. By offering a standardized syntax and a comprehensive set of instructions, cQASM improves interoperability, portability, and abstraction [7, p. 1].

6.1 Features of cQASM

6.1.1 Case Insensitivity and Comments

cQASM is not case-sensitive, so instructions can be written in either upper or lower case. Comments, indicated by the "#" symbol, can be added to improve readability and documentation [7, p. 3].

6.1.2 Qubit Definition and Addressing

cQASM supports defining qubit registers, which are groups of qubits that can be addressed by their index. For instance, `qubits 2` defines a register of two qubits, accessed as `q[0]` and `q[1]`. cQASM also supports renaming qubits and measurement outcomes to improve code clarity [7, p. 3].

6.1.3 Quantum Gates

cQASM includes a comprehensive set of quantum gates, such as single-qubit gates like the Hadamard (`h q[0]`), multi-qubit gates like the CNOT (`cnot q[0],q[1]`), and controlled gates (`c-x b[0],q[2]`). These gates can be extended with binary controls for conditional operations [7, p. 4].

6.1.4 Measurements

Measurements in cQASM can be performed on individual qubits or entire registers. The language supports measuring qubits in different bases (X, Y, Z) and advanced measurement techniques like parity measurements for multi-qubit observables [7, p. 4].

6.1.5 Feedback and Display

cQASM allows for binary-controlled quantum gates, where measurement results can influence subsequent operations. The `display` command outputs measurement results, which comes in handy especially in simulation environments where detailed state information is available [7, p. 4].

6.1.6 Code Example: Creation of a Bell state [7, p. 3]

```
-----------------------------------------------
version 1.0
# define a quantum register of 2 qubits
qubits 2

# create a Bell pair via a Hadamard rotation
h q[0]
# followed by a CNOT gate
# q[0]: control qubit, q[1]: target qubit
cnot q[0],q[1]

# measure both qubits to test correlations
measure q[0]
measure q[1]
-----------------------------------------------
```

The Bell state is an example of quantum entanglement between two qubits, playing an important role in quantum communication and algorithms. Creating a Bell state involves the following steps:
First, the Hadamard gate H is applied to the first qubit (`q[0]`), which places it in a superpositional state. Next, the $CNOT$ gate is applied to both qubits, with the first qubit (`q[0]`) as the control and the second qubit (`q[1]`) as the target, by this the two qubits are entangled.
The resulting state is one of the four Bell states, which are the simplest forms of entanglement between two qubits.
Mathematically, this Bell state is expressed as:

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

Upon measurement, the qubits show perfect correlation: if one qubit is measured in the state $|0\rangle$, the other will also be in the state $|0\rangle$, and the same for the state $|1\rangle$. This property is crucial for many quantum information protocols [8, p. 16].

6.2 Special Features

cQASM includes advanced features to enhance its expressiveness and usability:

For instance, there are parallel gates that enable the execution of multiple gates simultaneously. Other noteworthy features are the Single Gate Multiple Qubits (SGMQ) instruction, whichs allows a single quantum operation to be applied to multiple qubits, and the sub-circuit definition and static loops that support the modularization of circuits into reusable sub-circuits and iteration over these sub-circuits.

cQASM also offers measurement averaging and expectation values which is the collection and averaging of measurements that are of greate use in many quantum algorithms and error correction protocols [7, p. 5].
cQASM aims to provide a standardized, hardware-independent language for representing quantum circuits. Its comprehensive syntax and features support the development, simulation, and execution of quantum algorithms. As the field of quantum computing advances, cQASM is expected to evolve, adding new functionalities and improving the integration between high-level quantum programming and hardware-specific execution [7, p. 6].

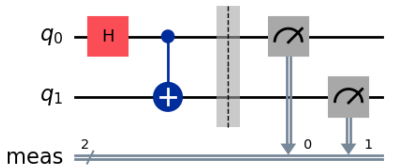
7 Qiskit

Qiskit is an open-source quantum computing framework developed by IBM. It provides tools for creating and manipulating quantum programs and running them on simulators or real quantum devices. Qiskit aims to make quantum computing accessible and practical for both researchers and developers by offering a comprehensive suite of libraries and modules [9].

7.1 Code Snippets Demonstrating Qiskit

Below are Python scripts demonstrating the capabilities of Qiskit. The code snippets create quantum circuits, execute them on a simulator, and visualize the results.

```python
from qiskit import QuantumCircuit

from qiskit_aer import Aer

import matplotlib.pyplot as plt

import numpy as np

backend = Aer.get_backend('qasm_simulator')

qc = QuantumCircuit(2)

qc.h(0)
qc.cx(0, 1)
qc.measure_all()

qc.draw('mpl')
plt.figure()

job = backend.run(qc, shots=1000)

result = job.result()

counts = result.get_counts(qc)
print(counts)
plt.bar(counts.keys(), counts.values())
plt.show()
```



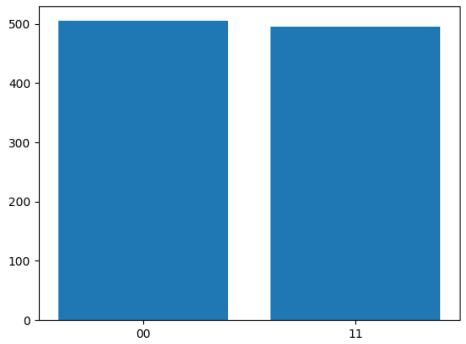
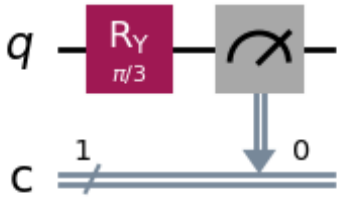


Figure 5: Snippet 1

In figure 7, the first image (on the left) displays the Python code used to create and run the quantum circuit. This code snippet imports necessary libraries, initializes the QASM simulator backend, defines a quantum circuit with two qubits, applies a Hadamard gate to the first qubit to create superposition, and uses a CNOT gate to entangle the qubits. The circuit is then measured, and the results are visualized using Matplotlib.
The second image (top right) visualizes the quantum circuit created by the code. The circuit consists of two qubits, q_0 and q_1. A Hadamard gate H is applied to the first qubit q_0, putting it in a superposition state. A CNOT gate follows, entangling q_0 with q_1. Finally, both qubits are measured, as indicated by the measurement symbols.
The third image (bottom right) displays the histogram of the measurement results after running the quantum circuit 1000 times. The histogram shows the distribution of the measurement outcomes 00 and 11, indicating that the qubits were found in these states with equal probability. This result is expected for an entangled state created by a Hadamard and CNOT gate sequence.

```
qc = QuantumCircuit(1, 1)

qc.ry(np.pi/3, 0)

qc.measure(0, 0)

qc.draw('mpl')
plt.figure()

job = backend.run(qc, shots=1000)

result = job.result()
counts = result.get_counts(qc)
print(counts)
plt.bar(counts.keys(), counts.values())
plt.show()
```



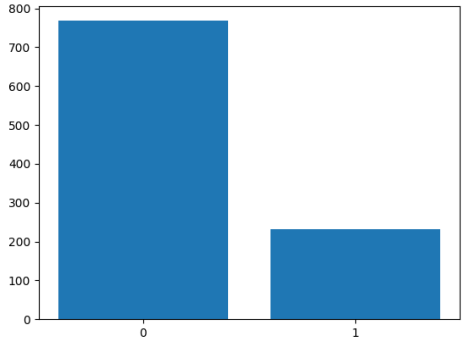


Figure 6: Snippet 2

In figure 6, a rotation $R_y(\pi/3)$ is applied to the qubit. This rotation transforms the qubit's state, creating a superposition where the amplitudes are determined by $\theta = \pi/3$. The resulting state is:

$$|\psi\rangle = R_y(\pi/3)\,|0\rangle = \cos(\pi/6)\,|0\rangle + \sin(\pi/6)\,|1\rangle$$

Given $\cos(\pi/6) = \sqrt{3}/2$ and $\sin(\pi/6) = 1/2$, the state becomes:

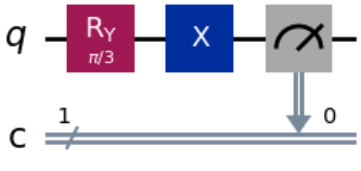
$$|\psi\rangle = \frac{\sqrt{3}}{2}\,|0\rangle + \frac{1}{2}\,|1\rangle$$

The probabilities of measuring the qubit in state 0 or 1 are the squares of the amplitudes:

$$P(0) = \left(\frac{1}{2}\right)^2 = \frac{1}{4}, \quad P(1) = \left(\frac{\sqrt{3}}{2}\right)^2 = \frac{3}{4}$$

After the measurement, these probabilities are displayed in the histogram, showing approximately 75% in state 0 and 25% in state 1.

```python
qc = QuantumCircuit(1, 1)

qc.ry(np.pi/3, 0)
qc.x(0)

qc.measure(0, 0)

qc.draw('mpl')
plt.figure()

job = backend.run(qc, shots=1000)

result = job.result()
counts = result.get_counts(qc)

print(counts)
plt.bar(counts.keys(), counts.values())
plt.show()
```



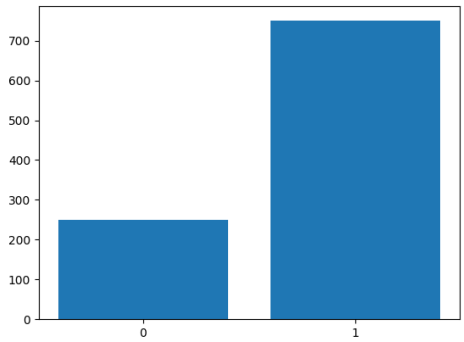


Figure 7: Snippet 3

In Snippet 3, a rotation $R_y(\pi/3)$ is also applied to the qubit, but an additional X gate is applied afterward. The X gate flips the qubit state, swapping the probabilities. After the rotation $R_y(\pi/3)$, the qubit is in the state:

$$|\psi\rangle = \frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle$$

Applying the X gate (which flips $|0\rangle$ to $|1\rangle$ and vice versa):

$$X|\psi\rangle = \frac{\sqrt{3}}{2}|1\rangle + \frac{1}{2}|0\rangle$$

Now, the probabilities are:

$$P(0) = \left(\frac{1}{2}\right)^2 = \frac{1}{4}, \quad P(1) = \left(\frac{\sqrt{3}}{2}\right)^2 = \frac{3}{4}$$

The histogram will show approximately 25% in state 0 and 75% in state 1. This result is the inverse of Snippet 2, reflecting the impact of the X gate on the rotation-induced superposition state.

8 Conclusion

An overview of quantum programming languages was provided, with a focus on their theoretical foundations and practical applications, and current state of development. The essential concepts of quantum computing, including qubits, superposition, and entanglement, were discussed in order to establish a foundation for understanding quantum algorithms.

A detailed examination was conducted of various quantum programming languages, with a particular focus on the Quantum Computation Language (QCL). The syntax, structure, and features of the language were highlighted to demonstrate how they are used to develop quantum programs. The combination of classical and quantum programming elements within QCL renders it a versatile tool for implementing and simulating quantum algorithms.

The Common Quantum Assembly Language (cQASM) was also discussed. cQASM addresses the need for a unified, hardware-independent language, thereby promoting interoperability and reliability across different quantum computing platforms. Standardization is a crucial aspect for the future development and execution of quantum algorithms.

To illustrate practical applications in Qiskit, three example snippets were presented, demonstrating the construction, execution, and visualization of quantum circuits.

The development of quantum programming languages is of great importance for fully exploiting the capabilities of quantum computers. As quantum hardware continues to evolve, the necessity for robust, standardized programming languages and tools will become increasingly apparent.

References

[1] Bernhard Ömer. Quantum Programming in QCL, 2000.

[2] Balamurugan K S, Sivakami A, Mathankumar M, Yalla Jnan Devi Satya prasad, and Irfan Ahmad. Quantum computing basics, applications and future perspectives. *Journal of Molecular Structure*, 1308:137917, 2024.

[3] Richard Kueng. Introduction to quantum computing, 2023. Lecture script, Fall 2023.

[4] Cryo-cmos electronic control for scalable quantum computing: Invited - scientific figure on researchgate. https://www.researchgate.net/figure/Bloch-sphere-representation-of-a-qubit$_fig1_3$17573486. *Accessed* : $2024 - 05 - 23$.

[5] Donald A. Sofge. A survey of quantum programming languages: History, methods, and tools. In *Second International Conference on Quantum, Nano and Micro Technologies (ICQNM 2008)*, pages 66–71, 2008.

[6] Bernhard Ömer. A Procedural Formalism of Quantum Computing, 1998.

[7] N. Khammassi, G. G. Guerreschi, I. Ashraf, J. W. Hogaboam, C. G. Almudever, and K. Bertels. cQASM v1.0: Towards a Common Quantum Assembly Language, May 2018. arXiv:1805.09607 [quant-ph] version: 1.

[8] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*, volume 2. Cambridge university press Cambridge, 2001.

[9] IBM Quantum Documentation.